ST.A Document Template for Software Verification and Validation Plan (SVVP) with Test Specifications

ST.A 1 Purpose of the plan

Example - testing: To prescribe the approach, resources and schedule of the testing activities for a level of testing. To identify the items being tested, the testing tasks to be performed and the personnel responsible for each task.

ST.A.2 Reference documents

ST.A.3 Definitions

ST.A.4 Verification and validation overview

ST.A.4.1 Organisation

Describe SVV roles, responsibilities and reporting lines.

ST.A.4.2 Master schedule

Summarize when SVV activities will be done.

ST.A.4.3 Resources summary

Describe who and what is required to do SVV.

ST.A.4.4 Responsibilities

List the people who will perform the roles described in 4.1.

ST.A.4.5 Tools, techniques and methods

Describe the tools, techniques and methods used for SVV.

ST.A.5 Verification and validation of administrative procedures (for large projects only)

ST.A.5.1 Anomaly reporting and resolution

Describe or reference the problem reporting procedures, ie, what reporting procedures are followed when problems are encountered.
Define the problem severity levels, eg, 3 – highly critical, 2-critical and 1-not critical.

ST.A.5.2 Control procedures

Describe or reference how SVV outputs will be controlled, eg, proper labelling of test results.

ST.A.5.3 Standards, practices and conventions

Describe any external policies, directives and procedures.

ST.A.6 Verification and validation activities (for large projects only)

ST.A.6.1 Formal proofs
Show how the correctness of the phase outputs will be shown, eg, correctness review signature.

ST.A.6.2 Reviews
Describe the inspection, walkthrough, technical review and audit procedures, ie, for untestable requirements.

ST.A.7 Acceptance Test Specification (*Note: Same Structure for Systems, Integration and Unit Test Specifications*)

ST.A.7.1 Test Plan

a) Introduction
Summarize the software items and features to be tested.

b) Test items
List the items to be tested.

c) Features to be tested
List the features to be tested.

d) Features not to be tested
List the features not to be tested.

e) Approach
Outline how the tests will be done.

f) Item pass/fail criteria
Specify the criteria for passing or failing a test.

g) Suspension criteria and resumption requirements
Specify the criteria for stopping or resuming a test.

h) Test deliverables
List the items that must be delivered before testing starts.
List the items that must be delivered when testing ends.

i) Testing tasks
Describe the tasks needed to prepare for and carry out the tests.

j) Environmental needs

Describe the resources & facilities required of the test environment.

k) Responsibilities
Describe who will:
Authorize testing is ready to start;
Perform the tests;
Check the results;
Authorize testing is complete.

l) Staffing and training needs
Describe test staffing needs by skill level;
Identify training requirements for the necessary skills.

m) Schedule
Summarize when test activities will be done.

n) Risks and contingencies
Identify the high-risk assumptions of this plan;
Describe the contingency plan for each.

o) Approvals
Specify who must approve this plan.

ST.A.7.2 Test Designs (repeat for each test design ..)

a) Test design identifier
Give a unique identifier for the test design.

b) Features to be tested
List the features to be tested.

c) Approach refinements
Describe how the tests will be done.

d) Test case identification
List the specific test cases.

e) Feature pass/fail criteria
Specify the criteria for passing or failing a test.

ST.A.7.3 Test Case Specifications (repeat for each test case ..)

a) Test case identifier
Give a unique identifier for the test case.

b) Test items
List the items to be tested.

c) Input specifications
Describe the input for the test case.

d) Output specifications
Describe the output required from the test case.

e) Environmental needs
Describe the test environment.

f) Special procedural requirements
Describe any special constraints on this test.

g) Test case dependencies
List the test cases that must precede this test case.

ST.A.7.4 Test Procedures (repeat for each test case ..)

a) Test procedure identifier
Give a unique identifier for the test procedure.

b) Purpose
Describe the purpose of the procedure.
List the test cases this procedure executes.

c) Special requirements
Describe any special constraints on this test.

d) Procedure steps
Describe how to log, setup, start, proceed, measure, shut down, restart, stop, wrap the test, and how to handle contingencies.

ST.A.7.5 Test Reports (repeat for each execution of a test procedure ..)

a) Test report identifier
Give a unique identifier for the test report.

b) Description
List the items being tested.

c) Activity and event entries
Identify the test procedure;
Say when the test was done, who did it;
What happened (overall test results, eg, passed, passed with minor defects, failed with one or more major problems, aborted with reason);

Describe where the outputs of the test procedure are kept.

ST.B Sample Test Specifications

ST.C Document Template for Software Transfer (STD)

ST.C.1 Introduction

ST.C.1.1 Purpose of the document

ST.C.1.2 Scope of the software

ST.C.1.3 Definitions, acronyms and abbreviations

ST.C.1.4 References

ST.C.1.5 Overview of the document

ST.C.2 Installation Procedures

Describe how to get the software up and running on the target machine.

ST.C.3 Build Procedures

Describe how to build the software from source code.

ST.C.4 Configuration Item List

List all the deliverable configuration items. Each configuration item should be expanded to include details where appropriate, eg,
Media to be delivered;
Licence no.;
Serial no.:
Version no. etc.

ST.C.5 Acceptance Test Report Summary

For each acceptance test, give the:
User requirement identifier and summary
Test report identifier in the AcceptanceTest Reports
Test result summary

ST.C.6 Software Problem Reports (SPR)

List the SPRs raised during the Acceptance phase and their status.

ST.C.7 Software Change Requests (SCR)

List the SCRs raised during the Acceptance phase and their status.

ST.C.8 Software Modification Reports (SMR)

List the SMRs completed during the Acceptance phase.

ST.D Form Template for Reporting Software problems (SPR)

Originator:	SPR No.:
	Date:
1. Software Item Title:	
2. Software Item Version/Release No.:	
3. Priority: Critical/Urgent/Routine (underline choice)	
4. Problem Description:	
5. Description of Environment:	
6. Recommended Solution (include date):	
7. Review Decision: Close/Update/Action/Reject (underline choice)	
8. Attachments:	

ST.E System Test Approach Discussion

ST.E.1 System Test Planning

The first step in system testing is to construct a system test plan and document it in the SVVP (Appendix 6.1). This plan is defined in the analysis phase and should describe the scope, approach, resources and schedule of the intended system tests. The scope of system testing is to verify compliance with the system objectives stated during the analysis phase. System testing must continue until readiness for user acceptance can be demonstrated.

The amount of testing required is dictated by the need to cover all the software requirements. A test should be defined for every essential software requirement, and for every desirable requirement that has been implemented.

ST.E.2 System Test Design

The next step in system testing is system test design. This and subsequent steps are performed in the design phase, although system test design may be attempted in the analysis phase. System test designs should specify the details of the test approach for each software requirement specified, and identify the associated test cases and test procedures. The description of the test approach should state the types of tests necessary (e.g. function test, stress test etc).

Knowledge of the internal workings of the software should not be required for system testing, and so white-box tests should be avoided. Black-box and other types of test should be used wherever possible. When a test of a requirement is not possible, an alternative method of verification should be used (e.g. inspection), to qualify/quantify the acceptance.

System testing tools can often be used for problem investigation. Effort invested in producing efficient easy-to-use diagnostic tools at this stage of development is often worthwhile.

If an incremental delivery or evolutionary development approach is being used, system tests of each release of the system should include regression tests of software requirements verified in earlier releases.

There are several types of requirements, each of which needs a distinct test approach. The following subsections discuss possible approaches.

ST.E.2.1 Function tests

System test design should begin by designing black-box tests to verify each functional requirement. Working from the functional requirements, techniques such as decision tables, state-transition tables and error guessing are used to design function tests.

ST.E.2.2 Performance tests

Performance requirements should contain quantitative statements about system performance. They may be specified by stating the:

- worst case that is acceptable;
- nominal value, to be used for design;
- best case value, to show where growth potential is needed.

System test cases should be designed to verify:

- that all worst case performance targets have been met;
- that nominal performance targets are usually achieved;
- whether any best-case performance targets have been met.

In addition, stress tests (see Section 6.5.2.13) should be designed to measure the absolute limits of performance.

ST.E.2.3 Interface tests

System tests should be designed to verify conformance to external interface requirements. Simulators and other test tools will be necessary if the software cannot be tested in the operational environment.

Tools (not debuggers) should be provided to:

- convert data flows into a form readable by human operators;
- edit the contents of data stores.

ST.E.2.4 Operations tests

Operations tests include all tests of the user interface, man machine interface, or human computer interaction requirements. They also cover the logistical and organizational requirements. These are essential before the software is delivered to the users.

Operations tests should be designed to show up deficiencies in usability such as:

- instructions that are difficult to follow;

- screens that are difficult to read;
- commonly-used operations with too many steps;
- meaningless error messages.

The operational requirements may have defined the time required to learn and operate the software. Such requirements can be made the basis of straightforward tests. For example, a test of usability might be to measure the time an operator with average skills takes to learn how to restart the system.

Other kinds of tests may be run throughout the system-testing period, for example:
- do all warning messages have a red background?
- is there help on this command?

If there is a help system, every topic should be systematically inspected for accuracy and appropriateness.

Response times should normally be specified in the performance requirements (as opposed to operational requirements). Even so, system tests should verify that the response time is short enough to make the system usable.

ST.E.2.5 Resource tests

Requirements for the usage of resources such as CPU time, storage space and memory may have been set as software requirements. The best way to test for compliance to these requirements is to allocate these resources and no more, so that a failure occurs if a resource is exhausted. If this is not suitable (e.g. it is usually not possible to specify the maximum size of a particular file), alternative approaches are to:
- use a system monitoring tool to collect statistics on resource consumption;
- check directories for file space used.

ST.E.2.6 Security tests

Security tests should check that the system is protected against threats to confidentiality, integrity and availability.

Tests should be designed to verify that basic security mechanisms specified as software requirements have been provided, for example:
- password protection;
- resource locking.

Deliberate attempts to break the security mechanisms are an effective way of detecting security errors. Possible tests are attempts to:

- access the files of another user;
- break into the system authorization files;
- access a resource when it is locked;
- stop processes being run by other users.

Security problems can often arise when users are granted system privileges unnecessarily. The Software User Manual should clearly state the privileges required to run the software.

Experience of past security problems should be used to check new systems. Security loopholes often recur.

ST.E.2.7 Portability tests

Portability requirements may require the software to be run in a variety of environments. Attempts should be made to verify portability by running a representative selection of system tests in all the required environments. If this is not possible, indirect techniques may be attempted. For example if a program is supposed to run on two different platforms, a programming language standard (e.g. ANSI C) might be specified and a static analyser tool used to check conformance to the standard. Successfully executing the program on one platform and passing the static analysis checks might be adequate proof that the software will run on the other platform.

ST.E.2.8 Reliability tests

Reliability requirements should define the Mean Time Between Failure (MTBF) of the software. Separate MTBF values may have been specified for different parts of the software.

Reliability can be estimated from the software problems reported during system testing. Tests designed to measure the performance limits should be excluded from the counts, and test case failures should be categorised (e.g. critical, non-critical). The mean time between failures can then be estimated by dividing the system testing time by the number of critical failures.

ST.E.2.9 Maintainability tests

Maintainability requirements should define the Mean Time to Repair (MTTR) of the software. Separate MTTR values may have been specified for different parts of the software.

Maintainability should be estimated by averaging the difference between the dates of Software Problem Reports (SPRs) reporting critical failures that occur during system testing, and the corresponding Software Modification Reports (SMRs) reporting the completion of the repairs.

Maintainability requirements may have included restrictions on the size and complexity of modules, or even the use of the programming language. These should be tested by means of a static analysis tool. If a static analysis tool is not available, samples of the code should be manually inspected.

ST.E.2.10 Safety tests

Safety requirements may specify that the software must avoid injury to people, or damage to property, when it fails. Compliance to safety requirements can be tested by:

- deliberately causing problems under controlled conditions and observing the system behaviour (e.g. disconnecting the power during system operations);
- observing system behaviour when faults occur during tests. Simulators may have to be built to perform safety tests.

Safety analysis classifies events and states according to how much of a hazard they cause to people or property. Hazards may be catastrophic (i.e. life-threatening), critical, marginal or negligible. Safety requirements may identify functions whose failure may cause a catastrophic or critical hazard. Safety tests may require exhaustive testing of these functions to establish their reliability.

ST.E.2.11 Miscellaneous tests

Some software requirements may specify the need for:

- documentation (particularly the Software User Manual or SUM);
- verification;
- acceptance testing;
- quality, other than reliability, maintainability and safety.

It is usually not possible to test for compliance to these requirements, and they are normally verified by inspection.

ST.E.2.12 Regression tests

Regression testing is 'selective retesting of a system or component, to verify that modifications have not caused unintended effects, and that the system or component still complies with its specified requirements'

Regression tests should be performed before every release of the software in the implementation phase. If an incremental delivery or evolutionary development approach is being used, regression tests should be performed to verify that the capabilities of earlier releases are unchanged.

Traditionally, regression testing often requires much effort, increasing the cost of change and reducing its speed. Test tools that automate regression testing are now widely available and can greatly increase the speed and accuracy of regression testing. Careful selection of test cases also reduces the cost of regression testing, and increases its effectiveness.

ST.E.2.13 Stress tests

Stress tests 'evaluate a system or software component at or beyond the limits of its specified requirements'. The most common kind of stress test is to measure the maximum load the software can sustain for a time, for example the:

- maximum number of activities that can be supported simultaneously;
- maximum quantity of data that can be processed in a given time.

Another kind of stress test, sometimes called a 'volume test', exercises the software with an abnormally large quantity of input data. For example a compiler might be fed a source file with very many lines of code, or a database management system with a file containing very many records. Time is not of the essence in a volume test.

Most software has capacity limits. Testers should examine the software documentation for statements about the amount of input the software can accept, and design tests to check that the stated capacity is provided. In addition, testers should look for inputs that have no constraints on capacity, and design tests to check whether undocumented constraints do exist.

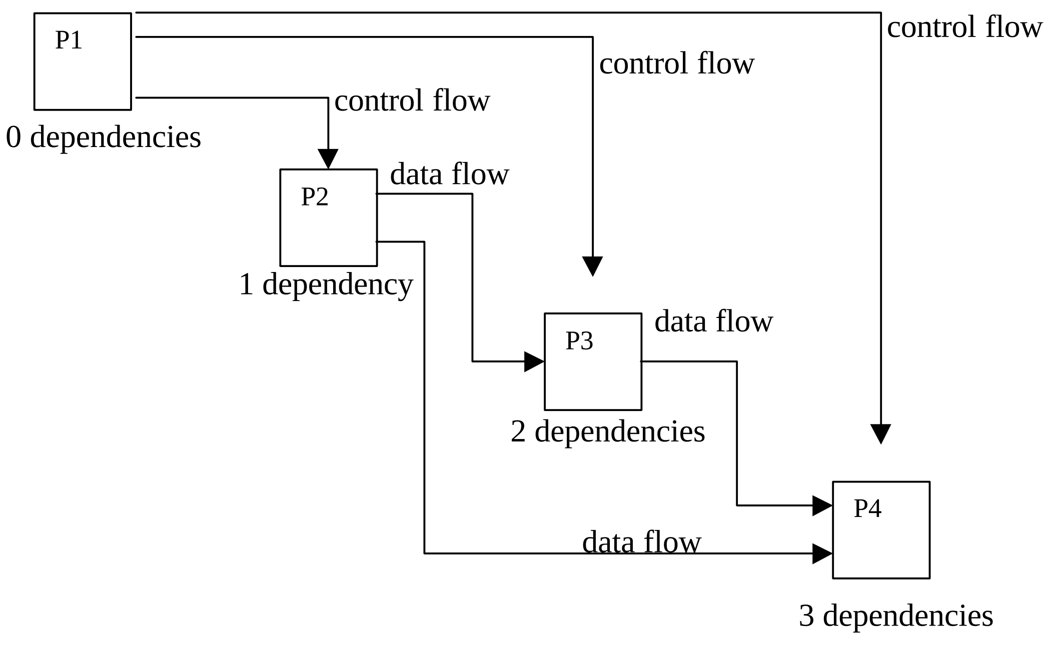
ST.F Example of How the Integration Sequence could be Determined

During integration test design performed at the design phase, details of the test approach for each software component defined during design are specified along with identification of the associated test cases and test procedures.

The description of the test approach should state the:
- integration sequence for constructing the system;
- types of tests necessary for individual components (e.g. white-box, black-box).

With the function-by-function method, the system grows during integration testing from the kernel units that depend upon a few other units, but are depended upon by many other units. The early availability of these kernel units eases subsequent testing.

For incremental delivery, the delivery plan will normally specify what functions are required in each delivery. Even so, the number of dependencies can be used to decide the order of integration of components in each delivery.



The above diagram shows a system composed of four programs P1, P2, P3 and P4. P1 is the 'program manager', providing the user interface and controlling the other programs. Program P2 supplies data to P3, and both P2 and P3 supply data to P4. User inputs are ignored. P1 has zero dependencies, P2 has one, P3 has two and P4 has three. The integration sequence is therefore P1, P2, P3 and then P4.

ST.G A Discussion of White-box, Black-box Integration and Performance Tests

ST.G.1 White-box Integration Tests

For file interfaces, test programs that print the contents of the files provide the visibility required. With real-time systems, facilities for trapping messages and copying them to a log file can be employed. Debuggers that set break points at interfaces can also be useful. When control or data flow traverses an interface where a break point is set, control is passed to the debugger, enabling inspection and logging of the flow.

The Structured Integration Testing method is the best known method for white-box integration testing. The integration complexity value gives the number of control flow paths that must be executed, and the 'design integration testing method' is used to define the control flow paths. The function-by-function integration method (see Appendix 6.6) can be used to define the order of testing the required control flow paths.

The addition of new components to a system often introduces new execution paths through it. Integration test design should identify paths suitable for testing and define test cases to check them. This type of path testing is sometimes called 'thread testing'. All new control flows should be tested.

ST.G.2 Black-box Integration Tests

Black-box integration tests should be used to fully exercise the functions of each component specified during design. Black-box tests may also be used to verify that data exchanged across an interface agree with the data structure specifications in the design.

ST.G.3 Performance Tests

During software design, resource constraints on the performance of a software unit may be imposed. For example a program may have to respond to user input within a specified elapsed time, or process a defined number of records within a specified CPU time, or occupy less than a specified amount of disk space or memory. Compliance with these constraints should be tested as directly as possible, for example by means of:

- performance analysis tools
- diagnostic code
- system monitoring tools.

ST.H Example of Incremental Assembly of Modules

The three rules of incremental assembly are:

- assemble the software units incrementally, module-by-module if possible, because problems that arise in a unit test are most likely to be related to the module that has just been added;
- introduce producer modules before consumer modules, because the former can provide control and data flows required by the latter.
- ensure that each step is reversible, so that rollback to a previous stage in the assembly is always possible, as a requirement of configuration management.

A simple example of unit test design is shown in Figure ST.H.1 The unit U1 is a major component of the software design. U1 is composed of modules M1, M2 and M3. Module M1 calls M2 and then M3, as shown by the structure chart. Two possible assembly sequences are shown. The sequence starting with M1 is 'top-down' and the sequence starting with M2 is 'bottom-up'. Figure ST.H.2 shows that data flows from M2 to M3 under the control of M1.

Each sequence in Figure ST.H.1 requires two test modules. The top-down sequence requires the two stub modules S2 and S3 to simulate M2 and M3. The bottom-up sequence requires the drivers D2 and D3 to simulate M1, because each driver simulates a different interface. If M1, M2 and M3 were tested individually before assembly, four drivers and stubs would be required. The incremental approach only requires two.

The rules of incremental assembly argue for top-down assembly instead of bottom-up because the top-down sequence introduces the:

- modules one-by-one;
- producer modules before consumer modules (i.e. M1 before M2 before M3).

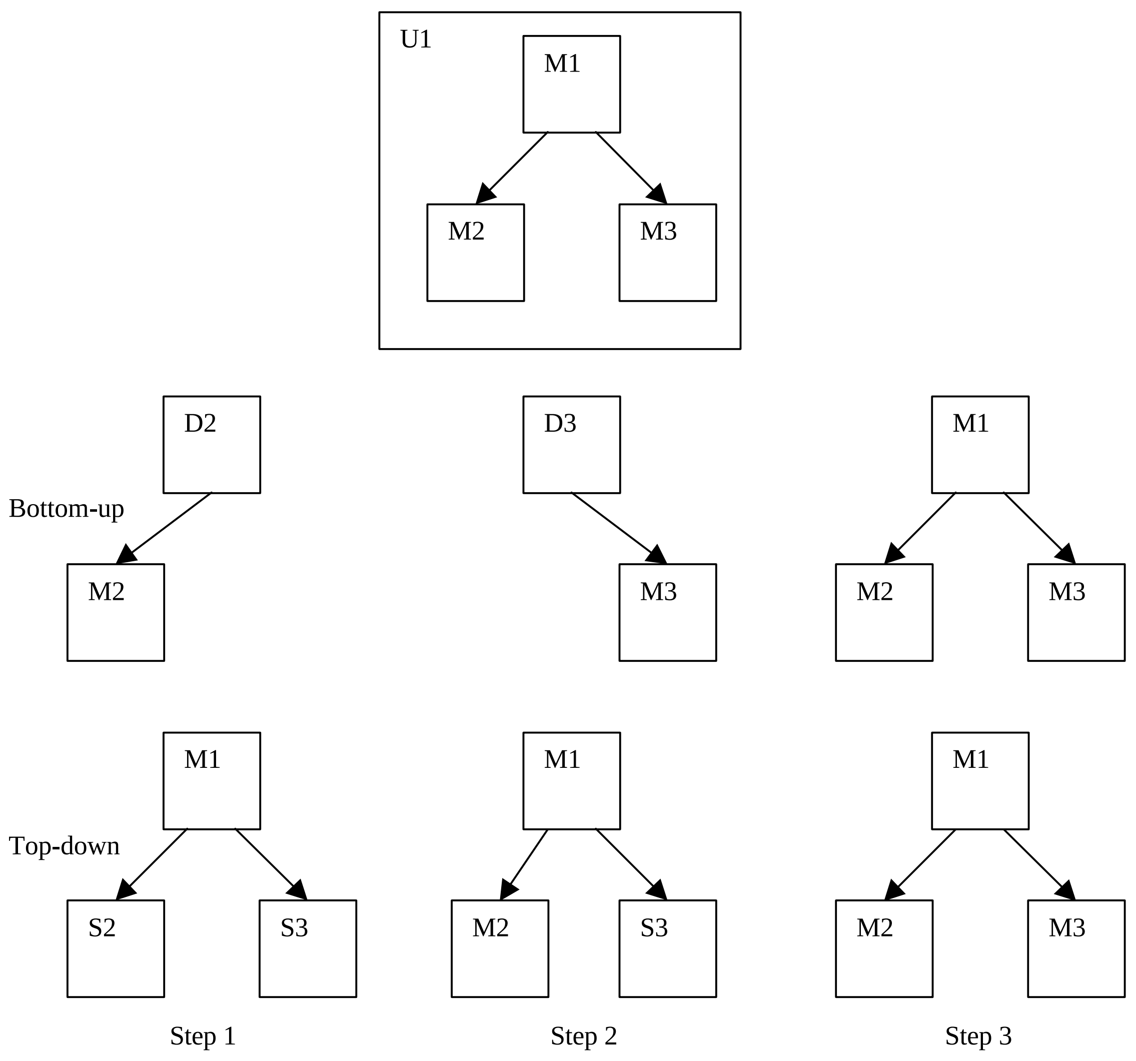


Figure ST.H.1 Example of unit test design

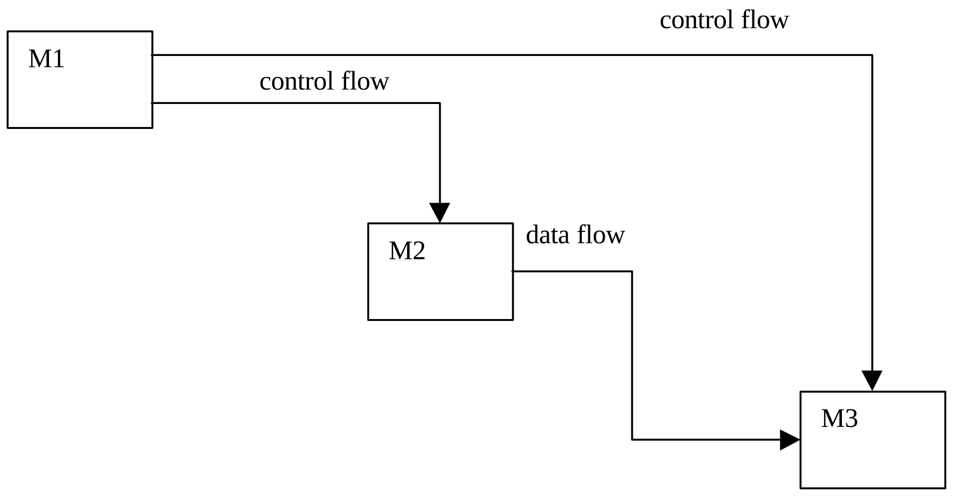


Figure ST.H.2 Data flow dependencies between the modules of U1

ST.I Unit Test Approaches

ST.I.1 White-box Unit Tests

The objective of white-box testing is to check the internal logic of the software. White-box tests are sometimes known as ‘path tests’, ‘structure tests’ or ‘logic tests’. A more appropriate title for this kind of test is ‘glass-box test’, as the engineer can see almost everything that the code is doing.

White-box unit tests are designed by examining the internal logic of each module and defining the input data sets that force the execution of different paths through the logic. Each input data set is a test case.

Traditionally, programmers used to insert diagnostic code to follow the internal processing (e.g. statements that print out the values of program variables during execution). Debugging tools that allow programmers to observe the execution of a program step-by-step in a screen display make the insertion of diagnostic code unnecessary, unless manual control of execution is not appropriate, such as when real-time code is tested.

When debugging tools are used for white-box testing, prior preparation of test cases and procedures is still necessary. The Structured Testing method is the best known method for white-box unit testing. The cyclomatic complexity value gives the number of paths that must be executed, and the ‘baseline method’ is used to define the paths. Lastly, input values are selected that will cause each path to be executed. This is called ’sensitising the path’.

A limitation of white-box testing is its inability to show missing logic. Black-box tests remedy this deficiency, if the tests are carefully planned to cascade into the missing logic of the white-box.

ST.I.2 Black-box Unit Tests

The objective of black-box tests is to verify the functionality of the software. The tester treats the module as ‘black-box’ whose internals cannot be seen. Black-box tests are sometimes called ‘function tests’.

Black-box unit tests are designed by examining the specification of each module and defining input data sets that will result in different behaviour (e.g. outputs). Each input data set is a test case.

Black-box tests should be designed to exercise the software for its whole range of inputs. Most software items will have many possible input data sets and using them all is impractical. Test designers should partition the range of possible inputs into ‘equivalence classes’. For any given error, input data sets in the same equivalence class will produce the same error.

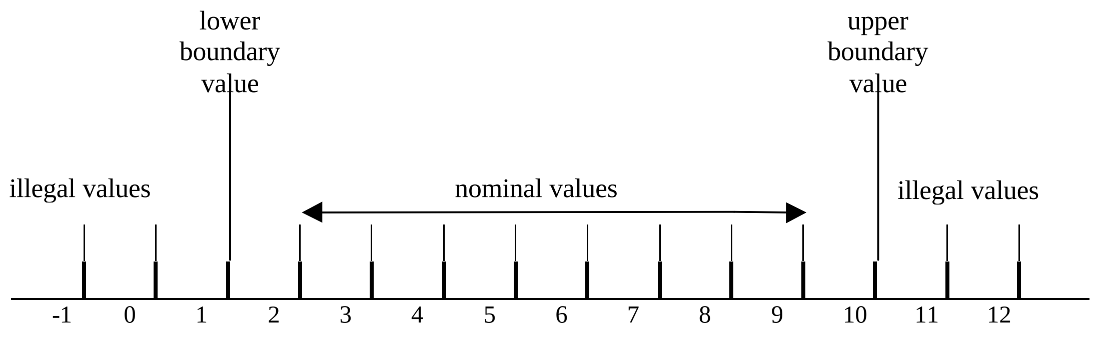


Figure ST.I.2.1 Equivalence partitioning example

Consider a module that accepts integers in the range 1 to 10 as input, for example. The input data can be partitioned into five equivalence classes as shown in Figure ST.I.2.1 The five equivalence classes are the illegal values below the lower boundary, such as 0, the lower boundary value 1, the nominal values 2 to 9, the upper boundary value 10, and the illegal values above the upper boundary, such as 11.

Output values can be used to generate additional equivalence classes. In the example above, if the output of the routine generated the result TRUE for input numbers less than or equal to 5 and FALSE for numbers greater than 5, the nominal value equivalence class should be split into two subclasses:

- nominal values giving a TRUE result, such as 3;
- boundary nominal value, i.e. 5;
- nominal values giving a FALSE result, such as 7.

Equivalence classes may be defined by considering all possible data types. For example the module above accepts integers only. Test cases could be devised using real, logical and character data.

Having defined the equivalence classes, the next step is to select suitable input values from each equivalence class. Input values close to the boundary values are normally selected because they are usually more effective in causing test failures (e.g. 11 might be expected to be more likely to produce a test failure than 99).

Although equivalence partitioning combined with boundary-value selection is a useful technique for generating efficient input data sets, it will not expose bugs linked to combinations of input data values. Techniques such as decision tables [Ref 9] and cause-effect graphs [Ref 11] can be very useful for defining tests that will expose such bugs.

	1	2	3	4
open_pressed	TRUE	TRUE	FALSE	FALSE
close_pressed	TRUE	FALSE	TRUE	FALSE
action	?	OPEN	CLOSE	?

Table ST.I.2.1: Decision table example

Table ST.I.2.1 shows the decision table for a module that has Boolean inputs that indicate whether the OPEN or CLOSE buttons of an elevator door have been pressed. When open_pressed is true and close_pressed is false, the action is OPEN. When close_pressed is true and open_pressed is false, the action is CLOSE. Table ST.I.2.1 shows that the outcomes for when open_pressed and close_pressed are both true and both false are undefined. Additional test cases setting open_pressed and close_pressed both true and then both false are likely to expose problems.

A useful technique for designing tests for real-time systems is the state-transition table. These tables define what messages can be processed in each state. For example, sending the message ‘open doors’ to an elevator in the state ‘moving’ should be rejected. Just as with decision tables, undefined outcomes shown by blank table entries make good candidates for testing.

Decision tables, cause-effect graphs and state-transition diagrams are just three of the many analysis techniques that can be employed for test design. After tests have been devised by means of these techniques, test designers should examine them to see whether additional tests are needed, their judgement being based upon their experience of similar systems or their involvement in the development of the system. This technique, called ‘error guessing’, should be risk-driven, focusing on the parts of the design that are novel or difficult to verify by other means, or where quality problems have occurred before.

Test tools that allow the automatic creation of drivers, stubs and test data sets help make black-box testing easier (see ST.H). Such tools can define equivalence classes based upon boundary values in the input, but the identification of more complex test cases requires knowledge on how the software should work.